# TileTrees

Sylvain Lefebvre, Carsten Dachsbacher

# TileTrees

Sylvain Lefebvre*
REVES / INRIA Sophia-Antipolis

Carsten Dachsbacher†
REVES / INRIA Sophia-Antipolis

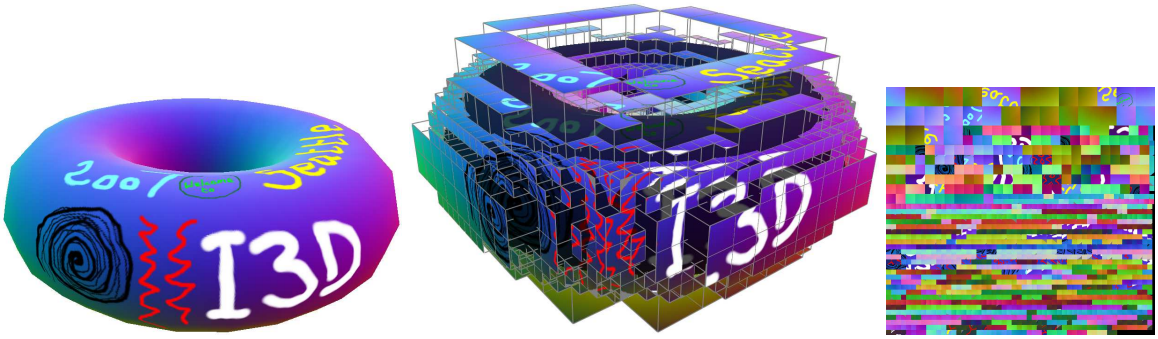

Figure 1: *Left:* A torus is textured by a TileTree. *Middle:* The TileTree positions square texture tiles around the surface using an octree. At rendering time, the surface is projected onto the tiles. *Right:* The tile map holding the set of square tiles.

## Abstract

Texture mapping with atlases suffer from several drawbacks: Wasted memory, seams, uniform resolution and no support of implicit surfaces. Texture mapping in a volume solves most of these issues, but unfortunately it induces an important space and time overhead.

To address this problem, we introduce the TileTree: A novel data structure for texture mapping surfaces. TileTrees store square texture tiles into the leaves of an octree surrounding the surface. At rendering time the surface is projected onto the tiles, and the color is retrieved by a simple 2D texture fetch into a tile map. This avoids the difficulties of global planar parameterizations while still mapping large pieces of surface to regular 2D textures. Our method is simple to implement, does not require long pre-processing time, nor any modification of the textured geometry. It is not limited to triangle meshes. The resulting texture has little distortion and is seamlessly interpolated over smooth surfaces. Our method natively supports adaptive resolution.

We show that TileTrees are more compact than other volume approaches, while providing fast access to the data. We also describe an interactive painting application, enabling to create, edit and render objects without having to convert between texture representations.

**Keywords:** texturing, texture mapping, interactive painting

## 1 Introduction

Texture mapping has become one of the fundamental component of computer graphics applications. By separating shape representation from surface appearance it provides a powerful and convenient tool to create, paint and render highly detailed objects and sceneries, at low geometrical cost.

The established approach for texture mapping is to flatten a surface into the 2D domain of a square image. The surface is *parameterized* into the plane. While some surfaces have a natural planar parameterization (cylinders, cones, more generally all developable surfaces), in most cases this operation is extremely difficult. Thus, the problem of finding good planar parameterizations has attracted lot of research interest and extremely powerful techniques have been devised [Floater and Hormann 2005]. The challenge is to flatten the surface while keeping a low distortion, which often requires to cut the surface in independently parameterized charts. These charts are later packed into a single texture atlas [Maillot et al. 1993]. Providing a tight packing is key to avoid wasting memory.

Nowadays, with the increasing demand for realism and high quality in computer graphics, artists are often required to paint detailed multi–layered texture maps for objects and large parts of the scenery. As a result, memory consumption for texture data increased much faster than available memory. The limitations and inherent difficulties of planar parameterization become less acceptable: Memory is wasted by the impossibility to efficiently pack irregular charts, or by oversampling to compensate for distortion. Most parameterizations are static and assume homogeneous sampling: Once computed they cannot adapt to artist painted content. Chart boundaries produce visible discontinuities in the bilinear interpolation. Finally, most methods encode the mapping as 2D coordinates in the vertices of a triangle mesh, and cannot be used on implicitly defined surfaces.

*e-mail: Sylvain.Lefebvre@sophia.inria.fr
†e-mail:Carsten.Dachsbacher@sophia.inria.fr

This lead several researchers toward adaptive and volume approaches. The object is immersed into a volume storing color information only around the surface. Data is stored in a spatial hierarchy, typically an octree. This solves a number of issues: A planar parameterization is no longer necessary, a seamless interpolation can be defined, the hierarchy can be refined in areas of interest. Moreover, the approach does not involve any complex pre-processing. Unfortunately, spatial hierarchies also imply an overhead in access time and space, which limits their use in interactive applications.

Our novel approach combines the advantages of volume approaches and 2D texture mapping. It removes the need for a global parameterization while relying on 2D textures for efficient packing and access. Our key idea is to use an octree to position square texture tiles around the surface. During rendering the surface is projected onto the tiles, and the color is retrieved by a simple 2D texture fetch into a tile map (see Figure 1). Note, however, that we do not seek to define a continuous texture domain: The tiles have different sizes and are packed together in arbitrary order. The square nature of the tiles makes efficient packing easier and let us define a seamless interpolation over the surface. Furthermore, by changing the resolution of the tiles we can locally and dynamically adapt the resolution to artist painted content.

## 2 Previous Work

Since its introduction by Catmull [Catmull 1974], texturing has spanned a large body of research. In particular, several researchers have focused on overcoming the limitations of texture mapping.

Importance driven parameterization methods have been proposed to allocate more resolution to regions with fine texture detail [Sloan et al. 1998; Sander et al. 2002; Balmelli et al. 2002]. Carr and Hart [Carr and Hart 2004] addressed the issue of dynamically updating the parameterization while the user paints on the surface. The geometry is clustered into charts mapped onto square regions of the texture. The resolution allocated to each chart is dynamically adapted to the user painted content. Our TileTree enables a similar interactive painting approach, but without the need to pre-cluster and dynamically parameterize the triangle mesh.

Several approaches avoid seams by parameterizing the surface onto regular charts [Purnomo et al. 2004; Carr et al. 2006]. While stored discontinuously, neighboring charts have corresponding samples: A continuous interpolation can be defined along the surface. To avoid splitting the geometry along chart boundaries, Tarini et al. [Tarini et al. 2004] parameterize surfaces on the faces of a regular polycube: A set of fixed size cubes surrounding the object. Not only does this define a continuous, tileable texture space, but the original mesh does not need to be modified. However, the polycube maps have some drawbacks: The fixed resolution has to be carefully chosen to match the geometric features, the construction requires manual intervention, and finally a triangle mesh is required to encode the parameterization.

To enable texturing of implicit surfaces and avoid explicit parameterization altogether, Benson et al. [Benson and Davis 2002] and DeBry et al. [DeBry et al. 2002] proposed to encode texture data in an octree surrounding the surface. This provides low distortion and adaptive texturing, at the expense of a space and time overhead: The tree contains many unused entries in its nodes, and accessing the data requires a long chain of indirections. Note that the number of indirections can be reduced at the expense of increased space overhead [Lefebvre et al. 2005; Lefohn et al. 2006]. Instead, Lefebvre and Hoppe [Lefebvre and Hoppe 2006] forgo adaptivity and compactly store fixed-resolution volume color data with a perfect spatial hash. The space overhead is very low and data is accessed with only two memory lookups. Unfortunately, both methods share difficulties inherent to volume approaches when it comes to interpolation.

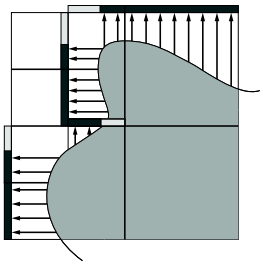

Figure 2: We position texture tiles around the surface using an octree. During rendering the surface is projected onto the tiles of closest orientation. The figure shows the 2D equivalent of a tile tree: A quadtree positioning 1D tiles around a curve.

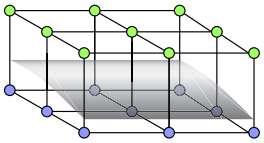

Figure 3: Correct tri-linear interpolation in a volume requires storing the surface as a thick layer. However, the bottom (blue) samples should be sufficient to texture the surface, leading to a 2x saving.

The surface is represented as a thick layer into the volume, thus requiring to store and access 8 samples (see Figure 3). However, interpolating over a surface should only require 4 samples: These approaches always store and access at least twice the data required to texture a given surface. In contrast, our TileTree maps 2D texture tiles onto the surface.

Finally, volume surface trees [Boubekeur et al. 2006] are also related to our work. The authors notice that subdividing an octree around a surface until very fine resolution is wasteful: After some level of subdivision the surface is faithfully captured by a simple heightfield. We follow a similar idea and stop subdividing the octree as soon as the surface can be textured by simple square tiles.

## 3 TileTrees

### 3.1 Overview

Our approach starts by building an octree around the surface to be textured, similarly to previous octree-based texturing methods. However - and this is the key idea of our work - instead of storing a single color value in the leaves, we map 2D *tiles* of texture data onto the *faces* of the leaves (up to six tiles per leaf). The tiles are compactly stored into a regular 2D texture, the *tile map*. During rendering, each surface point is projected onto one leaf face. This produces texture coordinates then used to access the corresponding texture tile. This idea is illustrated Figure 2.

We only subdivide the octree until no more than one fold exists in each leaf (see Section 3.2). Since with most geometry the texture detail is much finer than the geometric features, the octree leaves tend to be much larger than the texture pixel size: Many neighboring pixels share the same leaf, which guarantees a good access coherence.

The surface is projected onto the faces of the leaves with a simple parallel projection. The face to project onto is locally determined from the surface normal. Note that this projection only requires knowing the surface normal and enclosing leaf. It is performed dynamically *at rendering time*: Thus our approach does not require to store additional information in vertices. In fact, it does not even require vertices at all: It can be used on implicitly defined surfaces.

The following sections describe each aspect of our approach in more details.

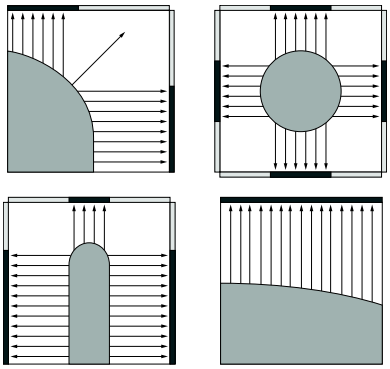

Figure 4: The normal to the surface is used to select on which face to project. The surface point is then mapped to the face with a simple parallel projection.

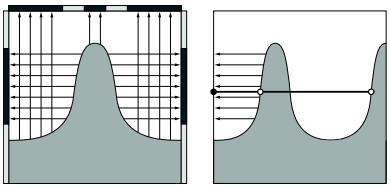

Figure 5: *Left:* Each leaf may contain up to one fold in the direction of projection. *Right:* If more than one fold is present the projection is no longer injective.

### 3.2 Projection

Each leaf of the octree encloses a piece of surface. Our goal is to project this part of the surface onto one or more faces of the leaf so that each point is uniquely textured. In other words the projection must be injective.

In addition, we want to perform the projection dynamically, at runtime. Therefore, it has to be as simple as possible to compute. However, if the projection fails to handle some surface configurations, we would have to subdivide the octree until reaching pixel resolution. As a compromise, we choose to perform a parallel projection onto the faces of the leaves. The face to project onto is chosen using the major direction of the normal to the surface. While being extremely simple to compute, this projection can handle successfully non trivial cases, including a full sphere. It also handles correctly two-sided thin surfaces, as illustrated Figure 4 – a difficult case with previous volume approaches [Benson and Davis 2002; DeBry et al. 2002]. In fact, it can even texture the back and front sides of a triangle differently.

However, it also has a few drawbacks. First, some small amount of distortion is present on steep surfaces. However, it is worth noting that this distortion is no greater than the one produced by an octree texture on faces at an angle. Second, it is not surjective: Some parts of the faces may never be covered by the projected surface. As each face is represented in memory by a square tile, this will result in wasted memory space (see Figure 6, left). We will see in the next section how this issue is addressed.

### 3.3 Building a tight octree

The octree surrounding the surface has to satisfy two constraints. First, we have to make sure to subdivide enough so that an injective projection is possible. Given the projection described in Section 3.2, this implies that no leaf must contain more than one fold in the directions used for projection (see Figure 5). Second, we seek to minimize memory waste. Consider a leaf in which the surface only projects partially onto the faces, leaving some unused pixels in the tiles. By further subdividing we get a better approximation of the surface and we increase pixel usage (see Figure 6). Of course, at the same time we increase the depth of the octree: There is a trade-off between space efficiency and octree complexity. We therefore expose a coverage threshold to the user. This threshold defines how acceptable it is to waste space to favor access efficiency. We also enforce a maximum tree depth, as this is mandatory for efficient GPU implementation [Lefebvre et al. 2005].

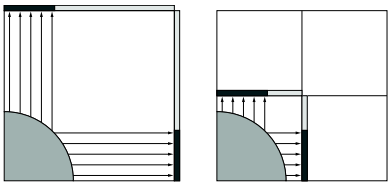

Figure 6: We enforce coverage by subdividing further leaves with coverage under a user specified threshold.

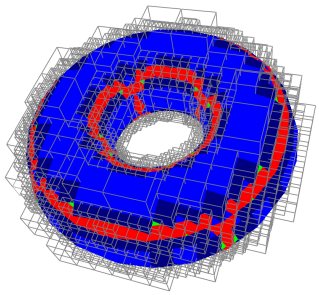

Figure 7: Leaves classification: *full-leaves* appear in blue, *stacked-leaves* in dark-blue, *boundary-leaves* in green and *n-leaves* in red.

Before describing the details of the octree construction, we define terms for the leaf configurations that occur. We distinguish two main types of leaves (see Figure 8). A *1-leaf* is a leaf where the surface projects onto a single face. A *n-leaf* is a leaf where the surface projects to more than one face. Now, within the set of 1-leaves we distinguish three subtypes. A *full-leaf* is a 1-leaf with a percentage coverage value of 1 (the tile is entirely used). We call a *leaf-stack* a set of 1-leaves at same octree level that are neighboring in the direction of the face supporting the tile (see Figure 8). The important property of a leaf-stack is that all the faces in the stack can share a same texture tile without overlapping. A leaf involved in a leaf-stack is named a *stacked-leaf*. Finally, a *boundary-leaf* is defined as a 1-leaf which is not involved in any leaf-stack and is not a full-leaf either. Figure 7 shows the various types of leaves on a 3D model.

In order to build the octree, we take the following steps, summarized in Figure 9:

1. Subdivide the octree until leaves contain one fold at most.
2. Enforce coverage constraint on n-leaves.
3. Enforce coverage constraint on boundary-leaves.
4. Split all 1-leaves that could form a stack with a neighbor at a deeper subdivision level.
5. Detect leaf-stacks (all members share a same texture tile).

Our implementation relies on ray–surface intersections for the tree construction, making it suitable for both polygonal meshes and implicit surfaces. Within each leaf we cast axis aligned rays to detect folds and obtain surface normals. The sampling rate must be high enough so that all leaf faces required for proper texturing will be detected, and so that no fold will be missed. Note that nothing precludes the use of more robust detection mechanisms.

Once the octree is created, we allocate tiles for each of the leaf face reached by the surface. The next section describes how tiles are allocated.

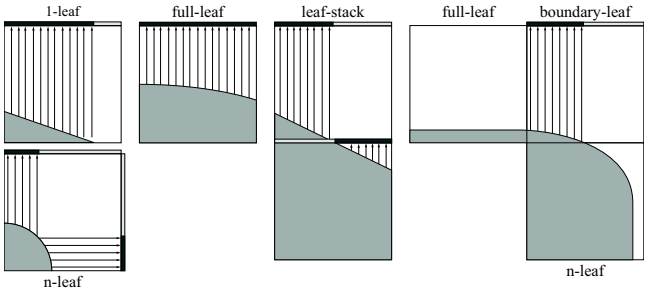


Figure 8: The 2 main types and 4 sub-types of TileTree leaves.

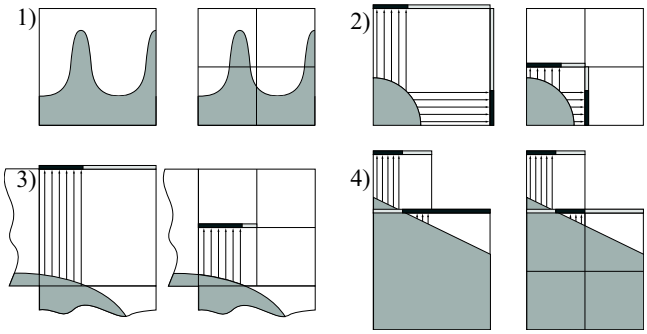


Figure 9: The octree construction involves four types of subdivisions. **1)** Split until no foldover. **2)** and **3)** Split to enforce coverage threshold. **4)** Split to create leaf-stacks.

### 3.4 Tiles

After octree subdivision, the number of tiles to allocate is:

$$\#\text{full-leaves} + \#\text{boundary-leaves} + \#\text{leaf-stacks} + \sum_{\{\text{n-leaf}\}} (n)$$

where # is the counting operator.

Each tile is mapped onto a leaf face so that the centers of the border samples are aligned with the boundaries of the face. This ensures seamless rendering and enables use of native hardware bilinear filtering. This also implies some sample duplication: Two neighboring tiles encode the same samples along their boundary. Using large tiles reduces this overhead.

To ensure matching samples between leaves at different subdivision levels, the tile size must be equal to $2^k + 1$, with $k$ a positive integer (see Figure 10, left). In case of adaptive resolution, this will also ensure that samples of the finer resolution tile are aligned with samples of the coarser resolution tile (see Figure 10, right).

For a simple access during rendering all tiles are packed together into a single texture: the tile-map. Tiles within n-leaves are stored contiguously. Thus for each leaf we allocate a texture rectangle with a size of $n(2^k+1) \times (2^k+1)$, $1 \leq n \leq 6$. The coordinates of the top left corners are stored in the octree leaves. The square shape of the tiles allows for simpler and therefore faster packing than with arbitrarily shaped charts. In our implementation we used a naive approach which places larger tiles first, from left-to-right and top-to-bottom. For speed-up, we track the first free column of every texture row. Although the packing can be further optimized by using quad-trees, it already performs very well, placing 18664 tiles in 96 milliseconds on the *armadillo* model of Figure 13.

### 3.5 Seamless interpolation

Interpolating the texture samples along the surface is key to avoid pixelated appearance. Seamless interpolation is a difficult issue which must not be overlooked. For instance, with volume approaches interpolation requires up to 8 lookups into the data structure [Lefebvre and Hoppe 2006], and more if adaptive texturing is used [Benson and Davis 2002]. Of course, this strongly impacts performance. This can be reduced by storing small blocks of data instead of point-wise colors, but border sample replication largely increases space overhead - especially in a volume where a thick layer has to be defined around the surface (see Figure 3).

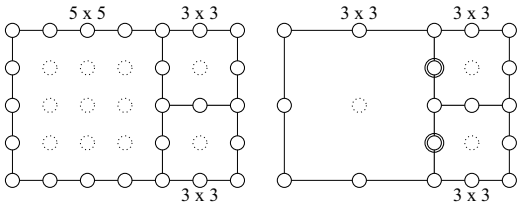


Figure 10: *Left:* For samples to match across boundaries, tiles must have a size of $2^k + 1$. *Right:* Samples of tiles at different resolution are aligned. Color at samples marked by two circles is computed to be the bi-linear interpolation of the neighbors. This ensure continuity across resolutions.

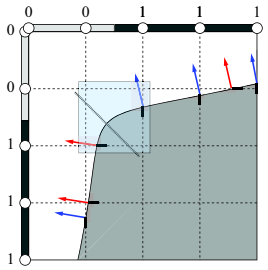


Figure 11: Within the outlined cell, the face used to texture the surface is abruptly changing, resulting in a visible seam. To perform interpolation within this cell, we flag whether each tile sample is used by the enclosed surface.

Our TileTrees enable fast seamless interpolation over smooth surfaces. We rely on the graphics hardware interpolation when accessing the tiles, and only have to perform a few more operations to obtain a seamless result. In the following discussion, we assume the per-sample color is already known. Please refer to Section 4 for an overview of how to fill a TileTree with texture content. Also, please note that seamless interpolation is currently limited to smooth surfaces. By *smooth surface*, it is to be understood that we refer to a *continuous normal field* over the surface, may it be a triangle mesh. We discuss this issue in more details in Section 6.

For correct interpolation, there are three cases to consider: full-leaves and stacked leaves, partially covered faces, and n-leaves interior.

**full-leaves**
Within full-leaves and stacked-leaves a correct seamless interpolation is guaranteed: The tile has all the necessary samples. Recall we have a one pixel border replication between neighboring tiles (see Section 3.4 and Figure 10, left).

**partially covered faces**
Within boundary-leaves and n-leaves the surface often only partially covers a face. Therefore, some tile samples have no defined color. At boundaries, these undefined colors would bleed-in during interpolation. Fortunately, the color of these samples is simply located in a tile of the neighboring leaf. For instance, consider the boundary-leaf of Figure 8. The missing samples are located in the n-leaf just below. We simply fill-in undefined samples by reading their color in the neighboring leaf.

**n-leaves interior**
N-leaves have an additional difficulty: The face accessed during rendering is abruptly changing along the surface (see Figure 11). The key idea to achieve a correct interpolation is to define the final color as a weighted sum of the contribution of all faces. Note that three faces can contribute at most: The three faces corresponding to the direction of the normal. We name the corresponding tiles $T_x$, $T_y$ and $T_z$, one for each main direction.

To define the interpolation weights, we store a binary flag into the alpha channel of the tile samples. This flag determines whether a sample is used by the enclosed surface (see Figure 11). This is easily computed by considering the surface normal at the location onto which the sample projects. Note that if the surface is not found under the sample, the color of the closest sample may be repeated. Alternatively, we can march along the faces to find the color of the sample on the neighboring tile. Note however that none of this will introduce a discontinuity: The final color is a *continuous* interpolation of the colors from all faces.

The flag is interpolated when accessing the tile data, so its value varies contiguously between $[0,1]$ on the surface. Within the n-leaf we now have three continuously varying flag values: $\alpha_x, \alpha_y$ and $\alpha_z$ interpolated from tiles $T_x$, $T_y$ and $T_z$. Note that if a face is not used by the surface it has no associated tile and we force its flag value to 0. The key idea is that the flag value will be 1 whenever the tile contains proper samples for the surface, and will continuously drop to 0 when the tile is no longer relevant. Interpolation only has to occur in areas where *none* of the flag values equal 1. This leads to a first definition of the weights $w_x$, $w_y$ and $w_z$:

$$\begin{aligned} w_x &= \alpha_x \cdot (1-\alpha_y) \cdot (1-\alpha_z) \\ w_y &= \alpha_y \cdot (1-\alpha_z) \cdot (1-\alpha_x) \\ w_z &= \alpha_z \cdot (1-\alpha_x) \cdot (1-\alpha_y) \end{aligned}$$

From which we compute the final color as:

$$c = \frac{(c_x \cdot w_x + c_y \cdot w_y + c_z \cdot w_z)}{(w_x + w_y + w_z)}$$

where $c_x$, $c_y$ and $c_z$ are colors fetched from tiles $T_x$, $T_y$ and $T_z$.

This works well in most cases, however these weights are not robust. If more than one flag equals 1, or all equal 0, the weights are null and the color is undefined. Both cases are possible if small geometric features are present in between tile samples. Fortunately, there exists a simple solution to this issue. We need to enforce that only one flag reaches 1 simultaneously. The normal to the surface gives exactly that: It always selects a single face, so we use it to dampen the flags. We thus compute the weights as:

$$\begin{aligned} damp &= abs(nrm)/max(|nrm_x|,|nrm_y|,|nrm_z|) \\ w_x &= \alpha_x damp_x \cdot (1-\alpha_y damp_y) \cdot (1-\alpha_z damp_z) \\ w_y &= \alpha_y damp_y \cdot (1-\alpha_z damp_z) \cdot (1-\alpha_x damp_x) \\ w_z &= \alpha_z damp_z \cdot (1-\alpha_x damp_x) \cdot (1-\alpha_y damp_y) \end{aligned}$$

where *nrm* is the normal to the surface at the point being considered, and *abs* a per-component absolute value. The zero case can then be avoided by always adding a small epsilon to the flag values. As long as the normal field is smooth, the weights are continuous and the final result is a seamless interpolation of the samples.

**adaptive resolution**
Whenever adaptive resolution is used, an additional difficulty appears: At the boundary between two tiles of different resolution, some samples of the higher resolution tile have no corresponding sample on the coarser resolution tile. This is illustrated Figure 10, right. With no specific treatment this produces obvious high-frequency discontinuities. In order to ensure smooth interpolation, we force the color of higher-resolution border samples to match the bi-linear interpolation of the lower-resolution samples.

Figure 12 illustrates seamless interpolation and adaptive resolution.


Figure 12: *Left:* Close-up on an n-leaf without interpolation. *Middle:* Same with seamless interpolation enabled. *Right:* Interpolation and adaptive resolution.

## 3.6 MIP-mapping

Due to the tile resolution of $2^k+1$, we cannot directly apply MIP-mapping to the tile map. MIP-mapping can be achieved by computing a separate tile map for each resolution level. This requires to store one tile coordinate per-level in the octree leaves. During rendering, the appropriate MIP-mapping level is computed and the color is fetched from the corresponding tile map. Two levels may be accessed for linear interpolation in-between MIP-mapping levels. Also note that in case of extreme undersampling, the tree itself may be MIP-mapped.

## 3.7 Implementation details

For the tree storage and lookup we rely on the hardware implementation of [Lefebvre et al. 2005]. Each leaf stores:

- A bit vector marking tile presence on each face (6 bits).
- The coordinates of the top left corner of the tiles within the tile map (two 16 bits numbers).
- The resolution of the tile (8 bits).

The complete pseudo-code for the shader is given below:

```
float4 tileTreeLooukp(float3 p) {
  /// lookup into the octree
  float4 leaf    = octree_lookup(p);
  /// decode faces presence
  float3 face_p  = decode_pos_face_presence(leaf);
  float3 face_n  = decode_neg_face_presence(leaf);
  /// coordinates within node
  float3 local   = frac(p * lvlsize);
  /// align samples on leaf boundaries
  float  tileres = decode_tile_resolution(leaf);
  float3 uvw     = local*((tileres-1.0)/(tileres))
                         +(0.5/(tileres));
  /// select faces
  float3 nrm     = normalize(IN.Nrm);
  v_p            = face_p *   nrm;
  v_n            = face_n * (-nrm);
  float3 id_p    = float3(X_P,Y_P,Z_P);
  float3 id_n    = float3(X_N,Y_N,Z_N);
  float3 faceid  = (v_p > 0) ? id_p : -1;
  faceid         = (v_n > 0) ? id_n : faceid;
  /// access tile data for present tiles
  float4 clr0=0,clr1=0,clr2=0;
  if (faceid.x>-1) clr0=tileLkup(leaf,faceid.x,uvw.yz);
  if (faceid.y>-1) clr1=tileLkup(leaf,faceid.y,uvw.xz);
  if (faceid.z>-1) clr2=tileLkup(leaf,faceid.z,uvw.xy);
  /// sample usage vector
  float3 alpha_xyz = 1e-6+float3(clr0.w,clr1.w,clr2.w);
  /// seamless interpolation
  float3 anrm    = abs(nrm);
  float3 damp    = anrm/max(anrm.x,max(anrm.y,anrm.z));
  alpha_xyz     *= damp;
  float3 inv     = (1-alpha_xyz);
  float3 w       = alpha_xyz * inv.yzx * inv.zxy;
  /// compute final color
  return (clr0*w.x+clr1*w.y+clr2*w.z)/(dot(w,1));
}
```

Figure 13: *Left:* Armadillo model textured with a uniform resolution of $1024^3$. The entire TileTree fits in 11.4 MB. *Right:* Dragon model textured with a uniform resolution of $1024^3$. The entire TileTree fits in 11.3 MB.

## 4 Filling with content

We introduced the TileTree data structure and explained how to access it during rendering. We now have to define means of filling a TileTree with texture content. In this section we describe an interactive painting tool for TileTrees, and briefly explain how to convert between TileTrees and other texture representations.

**Interactive painting**
Interactive painting is performed easily using TileTrees. The idea is simple: When building the tile map, we also create an auxiliary texture storing the coordinates of the surface points projecting to the tile samples. A simple ray-surface intersection is used for this purpose.

Once the auxiliary texture is built, painting is performed as a render to texture operation: We directly paint into the tile map using the GPU. We rasterize a quad covering the entire tile map, retrieve the world space coordinate of the samples from the auxiliary table, and check whether samples are inside the brush. If inside, their color is updated. Alpha blending is used to attenuate the brush influence. Painting is extremely fast, and the speed does not depend on the brush size: In fact all pixels may be updated at once if desired.

Painting is slightly more complex when adaptive resolution is used. Recall that for adaptive resolution we need to compute the color of some samples to obtain a smooth interpolation (see Section 3.5). In our current implementation, we update these samples after each paint stroke. Also, in our current application the user is in charge of increasing or decreasing the local tile resolution. A painting scheme automatically adapting the texture resolution, such as the one proposed by Carr and Hart [Carr and Hart 2004], could be easily designed on top of TileTrees. Please refer to the accompanying video for an example of an interactive painting session.

**Conversion**
Converting from an existing texture is a convenient feature. This can be used, for instance, to provide a basis for further painting onto an existing object. The key idea, similarly to interactive painting, is to rely on the coordinates of the surface points projecting to the samples. With a triangle mesh, it is easy to track texture coordinates and to fetch an initial color from an existing atlas. With a volume texture, the 3d coordinates can be directly used to obtain a color at each sample. Converting a TileTree back into a texture atlas can be performed similarly to the fast octree-atlas conversion described in [Lefebvre et al. 2005].

| | Memory size | Frame rate |
|---|---|---|
| TileTree | 11.4 MB | 91 FPS |
| Hashed texture<br>*8 lookups for tri-linear* | 15.7 MB | 34 FPS |
| Octree texture<br>*8 lookups for tri-linear* | 32.6 MB | 25 FPS |
| Hashed texture<br>*blocking for tri-linear* | 45.9 MB | 135 FPS |

Table 1: Comparison of a TileTree with octree texture and hashed textures on the *armadillo* model with an equivalent volume texture resolution of $1024^3$. Frame rate is measured with the viewpoint of Figure 13.

## 5 Results and Discussion

We compare TileTrees with other volume texture mapping approaches: octree textures and hashed textures. Table 1 summaries memory size and performance for each approach. TileTrees are slower than blocked hashed textures, but require 3 times less memory. They are, however, faster than hashed textures with 8 lookups for tri-linear interpolation, while still using less memory. It is remarkable that even with the space overhead of border sample duplication and memory waste of partially covered tile, TileTrees are still smaller than tri-linear hashed textures. This is of course due to the fact that volume texture mapping requires much more samples for interpolation. Note that for a fair comparison we did not use the adaptive capability of the TileTree, which would have further reduce memory usage.

Table 5 shows how octree complexity and tile map usage are linked. We fix the final resolution to $1024^3$ and vary the maximum octree depth. As the depth increases, we get a better usage of the tile map, but it reduces performance as the octree access gets more expansive.

Table 3 shows how the TileTree size evolves with the user adjustable coverage threshold. A higher coverage constraint decreases tile map size, but also increases the octree size. The optimal is therefore obtained for an intermediate value. In practice we often use a coverage threshold of 1 for compactness.

| Tree max depth | Octree size | Tile map size | Tile map usage | Frame rate |
|---|---|---|---|---|
| 5 | 27 KB | 17.4 MB | 49% | 110 FPS |
| 6 | 97 KB | 13.5 MB | 59% | 98 FPS |
| 7 | 303 KB | 11.1 MB | 68% | 93 FPS |
| 8 | 831 KB | 10.9 MB | 75% | 91 FPS |

Table 2: TileTree behavior for varying maximum tree depth and fixed resolution of $1024^3$.

| Coverage threshold | Number of tiles | Octree size | Tile map size | Tile map usage |
|---|---|---|---|---|
| 0 | 13802 | 103 KB | 18.12 MB | 52% |
| 0.25 | 20520 | 217 KB | 11.5 MB | 63.8% |
| 0.5 | 24423 | 260 KB | 11.0 MB | 66.5% |
| 0.75 | 27373 | 288 KB | 11.0 MB | 67.5% |
| 1 | 29185 | 303 KB | 11.1 MB | 67.6% |

Table 3: TileTree behavior for a fixed resolution of $1024^3$ with varying coverage threshold. Maximum tree depth is set to 7.

## 6 Limitations and Future work

TileTrees have two main limitations.

First, seamless interpolation fails where normals are not continuous along the surface. Typically, edges of a cube will not have a seamless interpolation. While this may be acceptable if the surfaces have different materials, it may be a problem if a continuous texture is desired across these edges. One possible direction to circumvent this issue is to use a smoothed normal for computing the interpolation weights. However, to properly take into account triangles at angles greater than 90 degrees, interpolation must now consider all faces, including those in a direction opposite to the normal. This complicates the TileTree access.

Second, and this may be the most important limitation, the normal field used for access has to be consistent with the real geometry. In particular, if normals have a large angle compared to the real surface, a very high distortion will result: The TileTree is mistaken in using a face that does not match surface orientation. This problem, however, only exists on triangle meshes with extremely coarse tessellations. In this situation a discontinuous normal may be used along the edge, removing distortion at the expense of the aforementioned seam in the interpolation.

Finally, as additional future work we would like to explore the opportunity of using TileTrees for efficient texture loading and caching.

## 7 Conclusion

We introduced the TileTree, a new data structure for efficient texture mapping. TileTrees store square texture tiles in the leaves of an octree. The surface is projected onto the tiles during rendering. The resulting texture is seamlessly interpolated along smooth regions of the surface, with little memory and access overhead.

We showed that TileTrees are more compact in memory than other volume approaches, while offering many of their advantages. A low distortion texture mapping is achieved. No parameterization needs to be explicitly stored, making the approach available for implicitly defined surfaces. Moreover, TileTrees natively support adaptive resolution, at no additional rendering cost.

We hope that TileTrees will provide artists and developers with a practical tool to texture map surfaces with as little difficulty as possible.

## Acknowledgements

Thanks to Christian Eisenacher for carefuly proof-reading the paper and to George Drettakis for the video voice over.

## References

BALMELLI, L., TAUBIN, G., AND BERNARDINI, F. 2002. Space-optimized texture maps. In *Proceedings of the Eurographics Conference*, 411–420.

BENSON, D., AND DAVIS, J. 2002. Octree textures. In *Proceedings of ACM SIGGRAPH*, ACM Press, ACM SIGGRAPH, 785–790.

BOUBEKEUR, T., HEIDRICH, W., GRANIER, X., AND SCHLICK, C. 2006. Volume-surface trees. *Proceedings of the Eurographics Conference 25*, 3, 399–406.

CARR, N. A., AND HART, J. C. 2004. Painting detail. *Proceedings of ACM SIGGRAPH 23*, 3, 845–852.

CARR, N., HOBEROCK, J., CRANE, K., AND HART, J. C. 2006. Rectangular multi-chart geometry images. In *Proceedings of the Eurographics Symposium on Geometry Processing*, ACM Press, 181–190.

CATMULL, E. E. 1974. *A Subdivision Algorithm for Computer Display of Curved Surfaces*. Ph.d. thesis, University of Utah.

DEBRY, D., GIBBS, J., PETTY, D. D., AND ROBINS, N. 2002. Painting and rendering textures on unparameterized models. In *Proceedings of ACM SIGGRAPH*, ACM Press, ACM SIGGRAPH, 763–768.

FLOATER, M. S., AND HORMANN, K. 2005. *Surface Parameterization: a Tutorial and Survey*.

LEFEBVRE, S., AND HOPPE, H. 2006. Perfect spatial hashing. *Proceedings of ACM SIGGRAPH 25*, 3, 579–588.

LEFEBVRE, S., HORNUS, S., AND NEYRET, F. 2005. *Octree Textures on the GPU*. Addison–Wesley.

LEFOHN, A., KNISS, J., STRZODKA, R., SENGUPTA, S., AND OWENS, J. 2006. Glift : Generic, efficient random-access gpu data structures. *ACM Transactions on Graphics 25*, 1, 60–99.

MAILLOT, J., YAHIA, H., AND VERROUST, A. 1993. Interactive texture mapping. In *Proceedings of ACM SIGGRAPH*, Computer Graphics Proceedings, Annual Conference Series, 27–34.

PURNOMO, B., COHEN, J. D., AND KUMAR, S. 2004. Seamless texture atlases. In *Proceedings of the Eurographics Symposium on Geometry Processing*, ACM Press, 65–74.

SANDER, P. V., GORTLER, S. J., SNYDER, J., AND HOPPE, H. 2002. Signal-specialized parameterization. In *Proceedings of the Eurographics Workshop on Rendering*, 87–100.

SLOAN, P.-P. J., WEINSTEIN, D. M., AND BREDERSON, J. D. 1998. Importance driven texture coordinate optimization. *Computer Graphics Forum 17*, 3, 97–104.

TARINI, M., HORMANN, K., CIGNONI, P., AND MONTANI, C. 2004. Polycubemaps. *ACM Transactions on Graphics 23*, 3 (Aug.), 853–860. Proceedings of ACM SIGGRAPH.